Message passing

- We've explored ways of synchronizing concurrent threads operating in the same address space. For different address spaces, we need something else.
- Distributed shared memory is convenient to program but care must be taken for performance.
- Message passing -> rendezvous, the remote procedure call
- There are several varieties of message passing:
  - Synchronous
  - Asynchronous
  - Capacity controlled
  - Conditional

Message passing: what

- We can pass: integers (ints), floating points (doubles), and objects (e.g. character strings).
- Within one machine space a pipe (buffer in common memory or a file) holds sent but not yet received messages.
- Between address spaces (between machines) a socket is used. It works like a pipe in that it buffers sent but not yet received messages.

Message passing: specifying recipients and originators

- Typical primitive operations executed by threads are:
  - Send(todestID, message)
  - Receive(fromID, message)
- Where “to” and “from” specify some place(s) to send/receive the message. Message is a reference to the object containing the message

Destination terminology

- “port”, “channel”, and “mailbox” are used to describe where messages are sent and where they are received.
- Instead of sharing variables, threads in different address spaces share message passing channels as implemented by the messaging system.

Channels

- What is a channel? A common reference point that both sender and receiver and mention as a source/destination for messages. Thus they can avoid referring to each other directly.
- Channels can be one-to-one, many-to-one (many senders share a single channel to talk to a receiver) one-to-many, many-to-many.
Sending messages and blocking

• Sending a message is analogous to a thread making a remote assignment of a value to a variable in another address space. If send is blocking, the sending thread delays until the remote assignment is made. In non-blocking sends, the sending thread does not delay.
• Asynchronous message passing is an extension or enhancement of semaphore P and V operations to include data.

Combinations of message sending

• Rendezvous: if send and receive are both blocking, then a simple rendezvous occurs when the message is transferred or copied from the sender to the receiver.
• Another choice is buffered and nonbuffered. A nonbuffered send blocks until a receiver executes a receive. A buffered send usually implies nonblocking, although there is an upper limit on the size of the buffer, at which point the send becomes blocking until some part of the buffer is freed.

Sending

• Blocking (waits for receiver or receiver is already waiting)
• Buffered, nonblocking: message is buffered but the receiver has not necessarily gotten it yet upon return from call to send. Designer’s choice: what to do if buffer is full?
• Nonbuffered, nonblocking, returns an error if no receiver is ready or waiting, returns okay if message is sent and received.

Receiving

• Blocking: waits for message, i.e., sender to send;
• Buffered, nonblocking: returns an error if no message is ready or waiting, returns okay otherwise.
• Nonbuffered, nonblocking: returns an error if no sender is ready or waiting, returns okay if message is sent and received.

Extended rendezvous

– An extended rendezvous (sometimes known as a remote procedure call) occurs if
  • The sender does a blocking send followed by a blocking receive
  • The receiver does a blocking receive followed by a blocking send

Extended rendezvous

Client (master)  server (worker)
Send(request)  receive(service request)
… (wait)  … do service request
receive(results)  send(results)
The difference between RPC and ER

- The difference between an extended rendezvous and an RPC is in the implementation.
  - The rendezvous is accepted by an explicit server thread, programmed by the designer of the server application; the thread waits in a loop for rendezvous request messages.
  - With RPC, the RPC package or library typically creates a new thread to execute the server procedure, transparently to the programmer. To the programmer, it's as if a server thread was there all along. But actually, it came into existence only due to the RPC.

Distributed mutual exclusion

- We can’t assume that the threads have any shared variables.
- Threads can block due to message passing, but none of the thread inter-communication methods (e.g. notify, wait, suspend, resume, etc.) work between processes.
- We can’t assume that messages arrive in the same order that they were sent.

An algorithm for distributed mutual exclusion

- We assume:
  - N nodes (N processes, or N computers)
  - error-free communication between all nodes
  - messages sometimes arrive in a different order than they were sent
  - nodes do not fail or halt at any time (nodes eventually respond to all query messages sent to them)

Basic idea of DME algorithm

Basic idea:
- do forever{
  - noncritical section code
  - choose a sequence number (one higher than those seen so far)
  - send it to all other nodes
  - wait for a reply message from all other nodes
  - enter critical section
  - postprotocol
  - reply to the deferreds if their sequence number was higher
}

DME algorithm is multi-threaded

- Each node has three threads:
  - one executing the “do forever loop”
  - One handles request from other nodes
  - Third waits for replies from all other nodes

What a node does

- Node i sends a reply or ack message to node j immediately if node j has a lower sequence number (higher priority) or if node i is not trying to enter its critical section
- Node i defers the reply (until node i gets into and then out of its critical section) if node j has a higher sequence number (lower priority) in its message. Ties are broken by node identifiers.
- A node chooses its sequence number by adding one to the highest sequence number it has seen so far in incoming messages from other nodes.
Why is this algorithm correct?

- A node does not enter its CS until it receives replied from all other nodes.
- There is no deadlock since ties are broken by node identifiers.
- There is not starvation in the absence of contention; if none of the other nodes wants to enter its critical section, replies are immediate.
- There is no starvation in the presence of contention; after a node exits its critical section it chooses a new sequence number the next time it wants to enter its critical section which will be higher than the other contending nodes (eventually).

Why the sequence numbers matter

- The sequence numbers is equivalent to time-stamping outgoing messages with the clock’s value. If the outgoing messages are not time-stamped, deadlock may result. If the arrival time of a message is used to determine when a remote node decided it wanted to enter its critical section, two nodes might send request-to-enter messages to each other at about the same time, and both think that they have priority.
- If time-stamped messages using a local clock without correction via sequence numbers (which have at least some feedback from the outside world), then starvation may occur. A node with a faster clock gets lower priority than the one with a slower clock and starves.

Mutual exclusion fails with no sequence numbers

1: B (at local time 20) sends a request to enter CS to A
2: B’s request received by A at A’s local time 10
3: A (at local time 11) sends an OK to B
4: A (at local time 12) sends a request to enter CS to B
5: B receives A’s request at B’s local time 30
6: B sends an OK to A at its local time 31
7: B receives A’s OK (sent as event #3) at its local time 35. (note that messages from A are being received out of order)
8: A receives B’s OK (at A’s local time 17)
9: A enters its CS
10: B enters its CS
   B thinks that A has entered and exited its CS at event #7

Why doesn’t this happen with sequence numbers?

1: B (at local time 20) sends a request to enter CS to A (seq. no. 5, say)
2: B’s request received by A at A’s local time 10
3: A (at local time 11) sends an OK to B
4: A (at local time 12) sends a request to enter CS to B (seq. no. 6)
5: B receives A’s request at B’s local time 30
6: B won’t send OK to A because it wants to enter and A’s seq. no. is higher than the one that B used.

Practical problems with this mutual exclusion algorithm

- Lots of messages required to get permission to enter a CS.
- Instead we could have a central server algorithm in which one nodes arbitrates critical section entries for all other nodes.
- Furthermore, this algorithm requires all nodes never to crash. With the central server algorithm, we only require that the central server node not go down.